Matrix Code

M.H. van Emden

Department of Computer Science
University of Victoria, Victoria, Canada
vanemden@cs.uvic.ca

arXiv:1109.5416v5 [cs.PL] 17 Jul 2012

Abstract

Matrix Code gives imperative programming a mathematical semantics and heuristic power comparable in quality to functional and logic programming. A program in matrix code is developed incrementally from a specification in pre/post-condition form. The computations of a code matrix are characterized by powers of the matrix when it is interpreted as a transformation in a space of vectors of logical conditions. Correctness of a code matrix is expressed in terms of a fixpoint of the transformation.

Categories and Subject Descriptors D.1.4 [*Programming Techniques*]: Sequential Programming; D.2.4 [*Software/Program Verification*]: Correctness Proofs; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control Structures; F.3.3 [*Studies of Program Constructs*]: Control Primitives

General Terms program verification, programming methodology

Keywords Floyd assertions, Hoare logic, verification-driven programming

1. Introduction

By *imperative programming* we will understand the writing of code in which the state of the computation is explicitly manipulated by assignments that change the value of a variable. As a programming paradigm, imperative programming should be compared, and contrasted, with functional and logic programming. Compared to these latter paradigms, imperative programming is in an unsatisfactory state. At least as a first approximation, a definition in functional or logic programming is both a specification and is executable. In imperative programming proving that a function body meets its specification is such a challenge that it is not considered part of a programmer's task. Another difference, probably related, is that functional and logic programming have an elegant mathematical semantics in which the behaviour of a definition is characterized as a fixpoint of the transformation associated with the definition.

This paper is a contribution to imperative programming in the form of a new language, called *Matrix Code*, in which programs take the form of a matrix with binary relations among states as entries. Matrix Code is distinguished by a development process that begins with a null code matrix, progresses with small, obvious steps, and ends with a matrix that is of a special form that is trivially translatable to a conventional language like Java or C. The result of the translation has the same behaviour as the one determined by the mathematical semantics of the code matrix. Therefore the latter can be said to be executable. As every stage in the development process is partially correct with respect to the specification (the correctness of the initial null code matrix is *very* partial) Matrix Code comes close to the ideal in which the code is itself a proof of partial correctness.

Plan of the paper In Section 2 we give a small example of the verification of imperative code by Hoare's method. We note certain features that point in the direction of Matrix Code. In Section 4 we define this language and show the same example translated to it. In Section 5 we explain how a code matrix is executed and we define its set of computations. In Section 6 we use the fact that a code matrix is not only an executable program but also a set of verification conditions to characterize partial correctness in terms of a fixpoint of the matrix. In Section 8 we solve the problem of Section 2 in the systematic manner that is unique to Matrix Code. The final two sections survey related work and draw conclusions.

2. Hoare's verification method

As an introduction to the verification method for imperative programming due to Hoare [10] we verify a Java version of the prime-number generating program developed by Dijkstra in [6]. The Java version of this program is shown in Figure 1.

We think of a computation as a sequence of *computation states* each of which consists of a *control state* (a code location) and a *data state* (a tuple of values of the variables)[1]. According to Hoare's method, conditions are attached to code locations. The conditions assert that certain relations between program variables hold at the code locations. When such a condition occurs in a loop, it is the familiar invariant of that loop. In Figure 1 we have indicated by the comments `S`, `A`, `B`, `C`, and `H` where these conditions have to be placed. Figure 2 contains the corresponding conditions.

The verification of the function as a whole relies on the verification of a number of implications defined in terms of conditions and program elements such as tests and statements. Consider Figure 1: because there is an execution path from `A` to `B`, one has to show the truth of

```
{A && k<N} j=p[k-1]+2; n=0; {B},
```

which has as meaning: if `A && k<N` (the *precondition*) is true and if

```
j=p[k-1]+2; n=0;
```

is executed, then `B` (the *postcondition*) is true. Because of the three elements: precondition, postcondition, and the item in between, this is called a *Hoare triple*. Figure 2 contains not only the conditions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'12, September 19–21, 2012, Leuven, Belgium.
Copyright © 2012 ACM 978-1-4503-1522-7/12/09. . . $10.00

[1] In this paper we consider only code that executes in a single activation record.

```
public static void primes(int[] p, int N) {
  // S
  int j,k,n;
  p[0] = 2; p[1] = 3; k = 2;
  // A
  while (k<N) {
    j = p[k-1]+2; n = 0;
    // B
    while (p[n]*p[n] <= j) {
      // C
      if (j%p[n+1] != 0) n++;
      else {j += 2; n = 0;}
    }
    p[k++] = j;
  }
  // H
}
```

Figure 1. A Java function for filling `p[0..N-1]` with the first `N` primes. At the points indicated by the comments S, A, B, C, H we need conditions to allow verification by Hoare's method. The identifiers and the structure are the same as in Dijkstra's example [6].

```
Conditions:
S: p[0..N-1] exists and N>1
H: p[0..N-1] are the first N primes
A: S && p[0..k-1] are the first k primes && k <= N
B: A && k<N && relB(p, k, n, j)
C: B && p[n]*p[n] <= j

relB(p,k,n,j)} means that there is no prime
between p[k-1] and j, and that j is not divided
by any prime in p[0..n], and that n<k.

Hoare triples:

{S} p[0]=2; p[1]=3; k=2; {A}
{A && k >= N}  {H}
{A && k < N} j=p[k-1]+2; n=0; {B}
{B && p[n]*p[n] <= j} {C}
{B && p[n]*p[n] > j} p[k++] = j {A}
{C && j%p[n+1] != 0} n++ {B}
{C && j%p[n+1] == 0} j += 2; n = 0 {B}
```

Figure 2. Conditions and Hoare triples for Figure 1. The meaning of a Hoare triple {A0} CODE {A1} is that if condition A0 is true and if CODE is executed with termination, then condition A1 is true.

for Figure 1, but also the set of verification conditions in the form of Hoare triples.

The term "condition" for the type of thing that occurs as precondition and postcondition in a Hoare triple is, in our view, rather compelling. However, it seems that in certain contexts "assertion" is a more natural alternative term. In this paper we will use both. At the same time, one should make a distinction between the condition as a linguistic expression and the set that is the meaning of that expression. We trust no confusion arises as we use "assertion" and "condition" interchangeably for both the expression and the meaning.

Figure 1 may seem to be the obvious, or even only, solution to the problem. But instructors in a beginners' programming course will see a wondrously creative variety of alternative solutions. Being solutions, they can all be verified by the same set of triples as in Figure 2. What all these solutions also have in common is the flow chart, and this flow chart is also verified by the same set of triples. In this sense the flowchart is a language-independent notation for an algorithm that also accommodates Hoare's verification method. In fact, the method originates with Floyd [8], who introduced it with flow charts. In spite of their merit of language-independence and verifiability we are not satisfied with flow charts because of their lack of heuristic power and because the lack of an attractive mathematical model. Flow charts are interesting because they are only a small step away from Matrix Code, which does have these two properties. We describe this step in the remainder of this section.

It would be tempting to say that, once we have a sufficient set of Hoare triples, we can forget the program in Figure 1: all information about it is in the Hoare triples of Figure 2. This may seem so because, for example, in

```
{A && k < N} j=p[k-1]+2; n=0; {B}
```

`A` stands for the condition defined earlier in that figure. What is missing is the fact that condition `A` is tied to code location `A`. We need the preconditions to be identified by a single letter that stands for a condition, so that all triples have the form {P}S{Q}.

We will show that an algorithm as set of triples of the form {P}S{Q} has an attractive mathematical model and has considerable heuristic power. Assuming then that all the information about an algorithm is in a set of items of the form {P}S{Q}, what is a convenient format for such a set? The most obvious seems a graph where the nodes represent conditions and where the directed edges are labeled with the middle items of the triples. Such a graph is also often used to represent a sparse matrix. A disadvantage of the matrix format is that it takes up an amount of space that is quadratic in the number of nodes. However the mathematical model that we propose for an algorithm as set of triples of the form {P}S{Q} is that of a transformation in a certain type of vector space of conditions. We are used to having such transformations represented by matrices rather than graphs. As in other uses of such transformations, matrix multiplication is a familiar and fundamental operation. For most people graph multiplication, though perfectly well defined, is not familiar. Hence we opt for the matrix representation and use Matrix Code as name for an algorithm as set of triples of the form {P}S{Q}.

But we are running ahead of the story: this is only relevant if we can get all triples in the form {P}S{Q}. We do this by generalizing the `S` from a statement to a binary relation between data states. Because of the essential role of binary relations we review and introduce the needed terminology and notation.

3. Preliminaries on binary relations

As binary relations are essential to Matrix Code we review notation and terminology. For the purposes of this paper, a binary relation R on a set D is a subset of the Cartesian product $D \times D$. If (s_0, s_1)

is in a binary relation, then we say that s_0 is an *input*; s_1 an *output* of the relation.

The null relation is the empty subset of $D \times D$. The identity relation I_D on D is $\{(s_0, s_1) \in D \times D : s_0 = s_1\}$. The union $R_0 \cup R_1$ of binary relations R_0 and R_1 is defined to be their union as subsets of $D \times D$. The composition $R_0; R_1$ of binary relations R_0 and R_1 is $\{(s_0, s_1) \in D \times D : \exists t \in D.\ (s_0, t) \in R_0 \wedge (t, s_1) \in R_1\}$. The inverse R^{-1} of a binary relation R is $\{(t, s) \in D \times D : (s, t) \in R\}$.

Let us call subsets of D *conditions*, anticipating their future use. The *left projection* of a binary relation R is defined as the condition $\{x \in D : \exists y \in D.\ (x, y) \in R\}$. Dually, the *right projection* of a binary relation R is defined as the condition $\{y \in D : \exists x \in D.\ (x, y) \in R\}$.

We generalize I_D to I_c, which means, for any condition $c \subseteq D$, by definition, $\{(x, x) \in D \times D : x \in c\}$. This induces a one-to-one relation between c and I_c:

$$x \in c \leftrightarrow (x, x) \in I_c.$$

Accordingly, at times we view a condition (alias assertion) as a subset of D; at times as a subset of I_D.

DEFINITION 1. *Given a condition $p \subseteq D$ and a binary relation $R \subseteq (D \times D)$, we write $\{\mathtt{p}\}\mathtt{R}$ for the right projection of $I(p); R$, where $I(p)$ is the binary relation $\{(x, x) \in D \times D : x \in p\}$.*

Hoare triples were intended to be applied to program statements. However, they have a natural interpretation for binary relations, as follows.

DEFINITION 2. *The Hoare triple $\{\mathtt{p}\}\mathtt{R}\{\mathtt{q}\}$ holds iff*

$$\{\mathtt{p}\}\mathtt{R} \subseteq \mathtt{q}.$$

DEFINITION 3. *A* trace *of a relation $R \subseteq (D \times D)$ is a possibly infinite sequence of elements of D such that for any pair $(s, s') \in (D \times D)$ such that s' follows s in the sequence we have that $(s, s') \in R$.*

A trace $s_0 \ldots s_{n-1}$ is closed *iff there is no $d \in D$ such that $(s_{n-1}, d) \in R$.*

A segment *$[\alpha, \omega]$ of a trace is a contiguous subsequence of the trace; $\alpha \in D$ is the first, $\omega \in D$ is the last element in the segment.*

4. Matrix code

As a first step toward matrix code we modify the nature of the middle term of the Hoare triple $\{\mathtt{P}\}\mathtt{T}\{\mathtt{Q}\}$. Conventionally `T` is a statement of a conventional language, typically changing the value of one or more variables.

Let us regard the collection of all variables accessible to the code as a tuple of the values indexed by the names of the variables. We call this tuple the *data state*. Thus, in Figure 1 the data state consists of the array `p` and the variables `k`, `n`, `j`.

The effect of a statement can be modeled as a binary relation on data states. Expressed in terms of sets, such a relation R is the set of pairs (x, y) such that $(x, y) \in R$ iff R's output y is a possible data state after executing the statement R, when R's input x is the data state before. This captures all terminating statements. In particular, an assignment statement `v = E` corresponds to the binary relation consisting of pairs (x, y) of data states where the `v` component of y is equal to the result of evaluating `E`, and all other components of y are equal to the corresponding ones in x.

We may have that R is not single-valued: there may exist x, y_0, and y_1 such that $(x, y_0) \in R$ and $(x, y_1) \in R$ and $y_0 \neq y_1$. That is, we admit nondeterministic statements, so y_0 is *a possible* output rather than *the* output. Modeling a statement as a relation R allows us to account for another computational phenomenon: it may be that for some x there is no y such that $(x, y) \in R$. This expresses the fact that for some data states as input the effect of the statement is not defined. For example, if the input data state x is such that in this data state $w = 0$, then for the relation modeling the statement $u := u/w$ there is no corresponding output y. But of course R can be a *function* on the set of data states so that it is defined for every state as input and that for each of these there is one and only one output.

We modeled the middle term `T`, which is conventionally a statement, in $\{\mathtt{P}\}\mathtt{T}\{\mathtt{Q}\}$ as a binary relation. We now generalize `T` by allowing it to be any binary relation over data states. We call this generalization of `T` a *transition*.

A transition may denote the empty relation. The identity relation on a set Δ of data states is $\{(s, s) : s \in \Delta\}$. A transition in the form of a boolean expression b denotes a subset of the identity relation. Such a transition we call a *guard*, following [7].

Thus guards can be composed with other transitions. If r_1 and r_2 are the meanings of transitions t_1 and t_2, then $r_1; r_2$ is the meaning of $t_1; t_2$, which is the transition consisting of the execution of t_1 followed by the execution of t_2. But either or both of t_1 and t_2 may be a guard, and then the effect of $t_1; t_2$ is equally well determined by the definition of composition of binary relations. The interpretation of transition elements allows the composition of a guard with any transition, whether that is a guard or not. For example, `v--; v > 0` and `v > 1; v--` are both well-defined transitions.

```
Conditions:
{S} p[0]=2; p[1]=3; k=2; {A}
{A} k >= N  {H}
{A} k < N; j=p[k-1]+2; n=0; {B}
{B} p[n]*p[n] <= j {C}
{B} p[n]*p[n] > j; p[k++] = j {A}
{C} j%p[n+1] != 0; n++ {B}
{C} j%p[n+1] == 0; j += 2; n = 0 {B}
```

Figure 3. Hoare triples for Figure 1. The middle terms in the verification conditions are transitions.

The purpose of the introduction of transitions is that we can write the verification conditions of Figure 2 as in Figure 3. We introduce a new programming language so that Figure 3 is itself a program and so that Figure 3 is also the verification of that program.

A natural notation of a set of items of the form $\{\mathtt{P}\}\mathtt{T}\{\mathtt{Q}\}$ is a matrix with rows and columns labeled by conditions. In this way the verification conditions of Figure 3 become the code matrix in Figure 12, a program in Matrix Code, the language. As we customarily do, the empty row of the start label and the empty column of the halt label have been omitted.

DEFINITION 4. *Given a set L of labels, a tuple of variables, boolean expressions testing a relation among the subset of these variables, and statements defined on a subset of these variables. A* code matrix *consists of an L-by-L matrix M, an L-indexed row vector of conditions preceding the sequence of rows of M, and an L-indexed column vector of conditions following the sequence of columns of M. For all i and j in L the element M_{ij} of column i and row j[2] is a* transition*, an expression denoting a binary relation. Among the labels there is one that labels an empty row; this is the* start label. *Among the labels there is one that labels an empty column; this is the* halt label.

[2] Note the transposition from the usual order. In this way the direction of execution is from i to j.

A code matrix is a way of writing a set of verification conditions, so has the status of a formula of logic. Yet it is also a program for a suitably defined abstract machine. This will be proved by defining the computations of a code matrix.

Matrix Code is a programming language that relies on an underlying base language in which to define the types of the variables and in which to write the statements and boolean expressions that make up the transitions. Matrix Code is defined informally, as done here. The fragments of base language that are needed are defined according to the standard of Java or C, as the case may be.

Some conventions for writing a code matrix: if a cell contains the null relation as a transition, then nothing is written in the cell; if a row or column is empty, it is omitted.

5. The computations of a code matrix

Section 4 gave a syntax of matrix code. We now define its operational semantics by defining execution of a code matrix.

DEFINITION 5. *A* computation state *is a pair (l, v) where l is a* control state *(in the form of a label) and v is a* data state *(in the form of a tuple of values of variables).*

Execution of a code matrix consists of the execution agent performing a sequence of cycles. The agent carries a computation state which is updated during a cycle. At the beginning of the cycle the agent carries state (l, v). It enters from the top of the matrix through the column labeled by l until it encounters a non-empty cell. Let r be the row in which this cell occurs and let R be the relation modeling the transition in this cell. If the data state v of the agent is such that there is a $(v, w) \in R$, then the agent exits to the right with computation state (r, w). This completes the cycle, and the agent begins a new cycle unless it exited through row `H`.

The agent may start a cycle in a column that does not contain a transition having its data state as input. In that case the agent does not complete the cycle.

Initially the agent carries the control state `S`. If and when the control state changes to `H`, execution halts with success.

DEFINITION 6. *The* binary relation associated with *a code matrix M with set Σ of computation states is the set of pairs $((l, v), (l', v')) \in \Sigma \times \Sigma$ such that $(v, v') \in M_{l,l'}$, the element in column l and row l' of M.*

A computation of M *is a trace of the binary relation associated with M.*

A computation is closed *if it is closed as a trace.*

See Figure 4 for an example of a computation.

```
N = 3

control | data
state   | state
        | k     j     n     p
     ---------------------------------
     S  | ?     ?     ?     {?,?,?}
     A  | 2     ?     ?     {2,3,?}
     B  | 2     5     0     {2,3,?}
     C  | 2     5     0     {2,3,?}
     B  | 2     5     1     {2,3,?}
     A  | 3     5     1     {2,3,5}
     H  | 3     5     1     {2,3,5}
```

Figure 4. Example of the trace for `N` equals 3 of the code matrix in Figure 12.

If a row is empty, then its label can only occur in the first state of any computation. Such a label is the start label. If a column is empty, then any computation state containing its label has to terminate the computation. Such a label is the halt label. In Figure 12, S is the start label and H is the halt label.

DEFINITION 7. *A computation is* successful *if it is closed and if its last computation state contains the halt label as control state; otherwise a closed computation is* failed*.*

DEFINITION 8. *Let M and N be code matrices with the same set L of labels and the same set Δ of data states. The* product *MN of M and N is a code matrix with L as set of labels and Δ as set of data states and with the cell $(MN)_{ik}$ in column i and row k containing $\bigcup_{j \in L} M_{ij}; N_{jk}$.*

Let I be the L-labeled matrix of binary relations over Δ that has the identity relation on Δ on the main diagonal and the empty relation elsewhere. Then we have $IM = MI = M$ with M any L-labeled matrix with binary relations over Δ as elements. We write M^n for $M^{n-1}M$ for a positive integer n while $M^0 = I$.

Matrix code can be viewed as a format for defining new binary relations in terms of the binary relations given by the statements and boolean expressions of the base language.

DEFINITION 9. *The* relation computed by *a code matrix M with start label S and halt label H is defined to be the set of (s, t) in $\Delta \times \Delta$ such that there exists a computation of M that starts with (S, s) and ends with (H, t).*

We characterize the relation computed by a code matrix in terms of its powers. First two lemmas concerning these powers.

LEMMA 1. *If a code matrix M has a computation containing a segment $[(l, s), (l', s')]$, then there exists an n such that $(s, s') \in (M^n)_{l,l'}$.*

Proof We proceed by induction on the segment length k. If $k = 1$ the computation has the form $(l, s), (l', s')$, so that (l', s') is the successor of (l, s). By Definition 6 we have $(s, s') \in M_{l,l'}$.

We assume the lemma true for k.
$(l, s), (l_1, s_1), \ldots, (l_{k-1}, s_{k-1}), (l', s')$ is a computation implies that there exists an n such that $(s, s_{k-1}) \in M^n_{l,l'}$ (by the induction hypothesis) and $(s_{k-1}, s') \in M_{l_{k-1},l'}$. By Definition 8 this implies that $(s, s') \in M^{n+1}_{l,l'}$.

LEMMA 2. *$(s, s') \in (M^n)_{l,l'}$ implies that there exists a segment $[(l, s), (l', s')]$ of a computation of M; this holds for all $n = 1, 2, \ldots$*

Proof We proceed by induction on n. $(s, s') \in M_{l,l'}$ implies that $[(l, s), (l', s')]$ is a segment. This takes care of the base case $n = 1$.

Assume the lemma for n.
$(s, s'') \in (M^{n+1})_{l,l''}$ implies that there exists an s' and an l' such that $(s, s') \in (M^n)_{l,l'}$ and $(s', s'') \in M_{l',l''}$ by Definition 8. Hence, by the induction assumption there exists a segment $[(l, s), (l', s')]$ and $(s', s'') \in M_{l',l''}$, which implies, by Definition 6, that there exists a segment $[(l, s), (l'', s'')]$ of a computation of M.

THEOREM 1. *Suppose that M is a code matrix with start state S and halt state H, with a finite set of labels, and a finite set of data states. Then the relation computed by M is $\bigcup_{i=0}^{\infty} (M^i)_{S,H}$.*

Proof Suppose that the pair (s, t) of data states is in the relation computation computed by M. By Definition 9 there exists a computation of M that begins with (S, s) and ends with (H, t).

According to Lemma 1 there is an n such that $(s,t) \in (M^n)_{S,H}$. Hence $(s,t) \in \bigcup_{n=0}^{\infty}(M^n)_{S,H}$.

Suppose that $(s,t) \in \bigcup_{n=0}^{\infty}(M^n)_{S,H}$. By the finiteness assumptions there exists an n such that $(s,t) \in (M^n)_{S,H}$. According to Lemma 2 this implies that there exists a computation of M that begins with (S,s) and ends with (H,t). Therefore (s,t) is in the relation computed by M, according to Definition 9.

6. Verification of matrix code

If the matrix in matrix code is in a certain relation with its row vector of preconditions and column vector of postconditions, then its computations are partially correct. In this section Theorem 2 makes this claim precise.

Conditions Transitions can be regarded as transformations of a single input to a single output. A transition can also be regarded as a *condition transformer*: transition T transforms condition p into the condition $\{\mathtt{p}\}\mathtt{T}$.

We characterize transitions by conditions of the form $\{\mathtt{p}\}\mathtt{T} \subseteq q$. According to Definition 2 this is written as $\{\mathtt{p}\}\mathtt{T}\{\mathtt{q}\}$. This notation was introduced with T as a binary relation in general. When T is the relation computed by a code matrix, this implies that condition p does not imply termination. Hence the correctness expressed by $\{\mathtt{p}\}\mathtt{T}\{\mathtt{q}\}$ is *partial* correctness.

In case the code matrix is nondeterministic there may be data states in p that begin computations that end in different data states; $\{\mathtt{p}\}\mathtt{T}\{\mathtt{q}\}$ implies that these final data states are all in q.

Condition vectors The transitions that are the elements of a code matrix define transformations on individual conditions. The matrix as a whole defines a transformation on condition *vectors*: vectors of conditions indexed by labels. The computations of a matrix have as elements computation states, which have the form (l,v), where l is a label and v is a data state. A set P of computation states defines a condition vector C by $C_l = \{v \in \Delta : (l,v) \in P\}$ for all $l \in L$. Conversely, C can be used to define $P = \bigcup_{l \in L}\{(l,v) : v \in C_l\}$. As the two correspondences are each others' inverse, condition vectors are isomorphic to sets of computation states.

DEFINITION 10. *The expression* $\{\mathtt{P}\}\mathtt{M}\{\mathtt{Q}\}$ *asserts that*

$$(\{\mathtt{P}\}\mathtt{M}) \subseteq Q,$$

where $\{\mathtt{P}\}\mathtt{M}$ *is the condition vector of which the* i*-th element is* $\bigcup_{j \in L} I(P_j); M_{ij}$*, for all* $i \in L$*. Here* $I(C)$ *is defined as the following subset of the identity relation on data states:* $\{(s,s) \in \Delta \times \Delta : C$ *is true in* $s\}$.

THEOREM 2. *Given a code matrix* M *and a condition vector* V *satisfying* $\{\mathtt{V}\}\mathtt{M}\{\mathtt{V}\}$*. For any computation state* (l',s') *of any computation beginning with* (l,s) *such that* $s \in V_l$ *it is the case that* $s' \in V_{l'}$.

Proof
We proceed by induction on the length n of the computation. If $n = 1$ (one state in the computation) we have $(l',s') = (l,s)$. Assume the theorem true for computations of length n. Consider the computation

$$(l,s),(l_1,s_1),\ldots,(l_{n-1},s_{n-1}),(l',s').$$

By the induction assumption $s_{n-1} \in V_{l_{n-1}}$. By Definition 6 $(s_{n-1},s') \in M_{l_{n-1},l'}$. It is given that $\{\mathtt{V}\}\mathtt{M}\{\mathtt{V}\}$, hence in particular that $\{\mathtt{V}_{\mathtt{l}_{\mathtt{n}-1}}\}\mathtt{M}_{\mathtt{l}_{\mathtt{n}-1},\mathtt{l}'}\{\mathtt{V}_{\mathtt{l}'}\}$. It follows that $s' \in V_{l'}$, which establishes the theorem for the computation of length $n+1$.

7. Mathematical semantics

A condition vector F such that $\{\mathtt{F}\}\mathtt{M} = F$ is a fixpoint of M when M is regarded as a transformer of condition vectors. Typically F is such that it has a compact description by means of boolean expressions. This is so because it derives from a program specification. But there is no reason to believe that this holds for $\{\mathtt{F}\}\mathtt{M}$. What makes Floyd's method useful is that it does not require finding a fixpoint of M, but only requires a solution V of $\{\mathtt{V}\}\mathtt{M} \subseteq V$ such that V_S is the condition at the start node and V_H the one at the halt node.

The fact that M, a monotonic transformation, is guaranteed, by the Knaster/Tarski theorem, to have a fixpoint is of no practical significance for two reasons. In the first place, we are not interested in verifying a given code matrix: the reason for using matrix code is that it helps us discover a program satisfying the specification. In the second place, the iterative algorithm that proves the existence of a fixpoint is not practically executable. The resulting fixpoint (the least such) is unlikely to have an intelligible description.

Condition vectors are an example of a *semimodule*, a generalization of the familiar vector space. In a vector space the scalars are elements of a field (e.g. the reals). If the scalars are generalized to elements of a ring (e.g. the integers), the vector space becomes a module. The analog of addition of integers is union of binary relations. Union does not have an inverse, so that we find that binary relations over a given domain are a semiring. The corresponding generalization of a vector is a semimodule. Thus the mathematical model of imperative programming obtained by Matrix Code is given by the theory of semimodules.

Semimodules are important in pure mathematics. For their role in programming we refer to Parker's monograph [12]. Parker defines a general framework, partial-order programming, which captures numerical optimization problems as well as functional and logic programming. As Parker shows, partial-order programming can take as special form semilinear partial-order programming, where the partially-ordered spaces take the form of semimodules. Examples of problems that find a natural formulation as transformations in semimodules expressible by matrices are: path reliability, path connectivity, maximum capacity paths, k-shortest paths, regular expressions, word abbreviations, path and cutset enumeration, and certain scheduling problems. In this paper we show that Matrix Code is semilinear partial-order programming, thereby inheriting a rich theory and sharing algebraic properties with many important applications.

8. Systematic program development

Floyd's method is difficult to apply because it is difficult to find the required conditions even when the program is correct. Because of this Dijkstra [4, 5] advocated parallel development of code and proof. In this section we demonstrate parallel development of a code matrix for the sample problem solved in Figure 1: to fill an array with the first N prime numbers in increasing order.

Background on prime numbers Before we start, let us review what we need to know about prime numbers. The following list of facts is not intended as a complete or nonredundant set of axioms; they are a selection to guide us in the choice of conditions and transitions.

1. *A prime is a positive integer that has no divisors.* (We do not count 1 or the integer itself as divisors. Moreover, 1 is not a prime.)
2. *There are infinitely many primes*, so the problem can be solved for any n.
3. *2 and 3 are the first two primes*. So a way to get started is to accept these as given and place them in the beginning of the

table. This has the advantage that we always have the situation where the last prime in the table is odd and the next odd number is the first candidate to be tested for the next prime.

4. *If a number has a divisor, then it has a prime divisor.* This can be used to save effort: we only have to test for divisibility by smaller primes, and these are already in the table.
5. *If a number has a divisor, then it has a prime divisor less than or equal to its square root.* This implies that we do not have to test the candidate for the next prime for divisibility by all primes already in the table.
6. *The square of every prime is greater than the next prime.* The significance of this fact will become apparent as we proceed.

Deriving the code matrix The distinctive advantage of matrix code is that a matrix can be expanded from the specification in small steps using only the *logic* of the application without needing to attend to the *control* component of the algorithm. Thus matrix code is an example of Kowalski's principle "Algorithm = Logic + Control" [11].

We assume that the specification exists in the form of a precondition and a postcondition. This gives rise to code matrix with one row and one column; the one in Figure 5.

`S: p[0..N-1] exists & N>1`	
`/*which T?*/`	`H: p[0..N-1] contains the first N primes`

Figure 5. There is only an empty transition T such that $\{S\}T\{H\}$.

The one element of this matrix is the transition `T` such that $\{S\}T\{H\}$ is true. That is, `T` has to be a simple combination of guards and assignment statements that places the `N` first primes in `p`, whatever `N` is. Absent such a `T`, we leave the matrix cell empty. The resulting code matrix satisfies $\{S\}T\{H\}$, which makes it partially correct, but *very* partially so: it has no successful computations. Although Figure 5 is the correct start of the development process, it is not the last step.

As it is too ambitious to place all primes in the array with a single transition, a reasonable thing to try is to fill it with the first `k` primes and then try to add the next prime after `p[k-1]`.

We need a condition `A` that is intermediate in the sense that $\{S\}T1\{A\}$ and $\{A\}T2\{H\}$ for simple `T1` and `T2`. Such a condition is: the first `k` primes in increasing order are in `p[0..k-1]` with `1 < k <= N`.

Condition `A` is promising because it is easy to think of such a `T1` and such a `T2`. The result is in Figure 6.

This again is a partially correct code matrix. It is a slight improvement in that it solves the problem if N happens to be one or two. In all other cases it leads to failed computations. The difficulty is that in column `A` we may have that `k < N`, so that we cannot make the transition to `H`. We need to find the next prime after `p[k-1]`. Let `j` be the current candidate for this next prime. That suggests for condition `B: A` is true and `j` is such that there is no prime greater than `p[k-1]` and less than `j`. Moreover, `j` is not divisible by any of `p[0..n]`. This condition is abbreviated to `relB(p,k,n,j)`. It is a useful condition, as there is a simple transition that makes this true.

In the new column `B` it is easy to detect whether `n` is large enough to conclude that `j` is the next prime after `p[k-1]`. We place the corresponding transition in column `B` and we have Figure 11.

`A:`	`S: p[0..N-1] exists & N>1`	
`k >= N`		`H: p[0..N-1] contains the first N primes`
	`p[0] = 2; p[1] = 3; k = 2`	`A: p[0..k-1] contains the first k primes & k <= N`

Figure 6. In column A the case `k < N` is missing.

There are still failed computations. (In fact, there is still no way to get beyond $N = 2$.) The way ahead is clear: a transition is missing in column `B`, for the situation where `n` is too small to conclude that `j` is the next prime. That in itself produces condition `C` and, with it, a new row and column.

In column `C` the missing information is whether `j`, the candidate for the next prime, is divisible by `p[n+1]`. If not, then `n` can be incremented, and condition `B` is verified. If so, then `j` is not a prime and the search for the next prime must be restarted with `j+2`. This determines a transition in column `C` that verifies condition `C`, so is placed in that row. See Figure 12.

Up till now we detected with every additional row and column that the new column lacked a transition. Not this time: none of the columns has a missing transition. The code matrix has no failed computations. So it gives the correct answer by exiting in row `H`, or it continues in an infinite computation. As we have only proved partial correctness, this latter alternative remains a possibility.

Termination For an infinite computation to arise, there must be at least one condition that is revisited an infinite number of times. For each condition we give a reason why it can only be revisited a finite number of times.

1. Condition `A`. For this condition to be returned to, `k` has to have increased.
2. Condition `B`. For this condition to be returned to, `n` or `j` has to have increased.
3. Condition `C`. For this condition to be returned to, `n` has to have increased.

The transitions have been chosen so that the corresponding revisiting condition is satisfied. As none of these conditions can be satisfied an infinite number of times, the code matrix has no infinite computation.

Running matrix code Running a code matrix in current practice requires translation to a currently available language. Our examples of matrix code have been constructed for ease of translation to languages like Java or `C`. This entails a drastic reduction in expressivity. Let us now demonstrate translation using Figure 12 as example.

As there is a similarity between the control states and the states of a finite-state automaton (FSA), a good starting point for systematic translation of a code matrix is the pattern according to which an FSA is implemented. This is usually done by introducing a constant for every state and to let a variable, say, `state` assume these constants as values. An infinite loop containing a `switch` controlled by `state` then contains a `case` statement for every control state. The fact that in a programming language the case statements are

not restricted to input or output is the generalization that produces a code matrix from an FSA.

Each column of a code matrix translates to a `case` statement. The order in which the translations of the columns occur does not matter as long as `state` is initialized at `S`. Here we have arbitrarily chosen alphabetic order. In this way Figure 12 translates to the following.

```
public static void primesCM(int[] p, int N) {
  final int S=0, A=1, B=2, C=3, H=4;
  int state=S;    // control state
  int j=0, k=0, n=0; // data state
  while (true) {
    switch (state) {
      case A:
        if (k >= N) state = H;
        else {j = p[k-1]+2; n = 0; state = B;}
      break;
      case B: if (p[n]*p[n] > j) {
                p[k++] = j; state = A;
              } else state = C;
      break;
      case C:
        if (j%p[n+1] != 0) {n++; state = B;}
        else {j += 2; n = 0; state = C;}
      break;
      case H: return;
      case S: p[0] = 2; p[1] = 3; k = 2; state = A;
      break;
    }
  }
}
```

Figure 7. Translation of the code matrix in Figure 12 to Java.

A transition `b0;S0` in column X and row R_0 and transition `!b0;S1` in column X and row R_1 translate to `case X: if (b0) {S0; state = R0;} else {S1; state = R1} break;` in the above code.

9. Expressiveness of matrix code

The Java code obtained by translating a code matrix is quite different from what one conventionally would write: compare Figure 1 with Figure 7. In this example Matrix Code has the advantage of being a verification and of being easy to discover. But in the prime-number problem Matrix Code does not lead to a more efficient program: it has the same set of computations as the conventional one.

In this section we present an example where Matrix Code makes it easy to discover an algorithm that is more efficient than what is obtained via the conventional programming style. Consider the merging of two monotonically nondecreasing input streams into a single output stream. We have available the following C++ functions.

```
bool getL(int& x);  // output parameter x
bool getR(int& x);  // output parameter x
void putL();
void putR();
```

where `getL` (`getR`) tests the left (right) input stream for emptiness. In case of nonemptiness the output parameter `x` gets the value of the first element of the stream. Neither `getL` nor `getR` change any of the streams. This is only done by the functions `putL()` and `putR()` which transfer the first element of a nonempty left or right input stream to the output stream.

Figure 8 is a typical program for this situation. It typically acts in two stages. In the first stage both input streams are nonempty. In the second stage one of the input streams is empty so that all that remains to be done is to copy the other stream to the output.

```
void eMerge() {
  int u,v;
  while (getL(u) && getR(v))
    if (u <= v) putL();
    else        putR();
  while (getL(u)) putL();
  while (getR(v)) putR();
}
```

Figure 8. A structured program for merging two streams.

This algorithm performs unnecessary tests: in the first stage only one of the input streams is changed, so that only that one needs to be tested for emptiness; here both are tested[3]. It is superfluous tests like this that allow the algorithm to be as simple as it is.

Of course it is unlikely that it is important to save the kind of test just mentioned. But there are many types of merging situations and there may be some in which it does matter. An advantage of matrix code is that it does not bias the programmer towards including superfluous tests.

We proceed to develop a code matrix for merging. The assertions need to indicate whether it is known that an input stream is empty and, if not, what its first element is. If an input stream is possibly empty then we represent it by "`?`". We write "`e`" if an input stream is empty. Nonemptiness is indicated by writing "`x:?`", where `x` is the first element. We have to do this for each of the input streams; we write e.g. the assertion `(u:?,v:?)` to mean that both input streams are nonempty and have first elements `u` and `v`, respectively.

With these conventions we can state the program's specification as obtaining a transition from the state `S`, which is `?:?`, to the state `H`, which is `e:e`, while maintaining the invariant that the result of appending the output stream to the result of merging the input streams is constant. Accordingly, the development starts with Figure 9.

`S:(?,?)`	
`/*which T?*/`	`H:(e,e)`

Figure 9. Matrix code corresponding to specification of the merging program. But there is no `T` such that {`S`}`T`{`H`}. The conditions in this figure, as well as those in Figures 13 and 14 include the unstated conjunct that the result of appending the output stream to the merge of the input streams is equal to the merge of the input streams in the initial state.

As always with matrix code, we start with the conditions. Which do we need, in addition to the `(?,?)` and `(e,e)` given by the specification? For each of the input streams there are three states of information:

[3] With the one exception when the left input stream runs out at the same time as, or before, the right input stream.

- `?`
- `e`
- `x:?` for some first element `x`

It is to be expected that the two input streams can assume each of the three information states independently, for a total of nine conditions.

It is desirable that the initial condition `(?,?)` of minimal information does not arise during a computation of the code matrix. Under the assumption that we can avoid this there will be only rows for the eight other conditions. By the time we will have populated the columns for these eight conditions we will see whether this assumption was justified.

This problem is easy because the conditions are determined by the nature of the problem. For each condition there is an obvious and easy-to-realize revisiting condition. If there is at least one unknown input stream at least one of them has to become known before revisiting. If both input streams are known, then at least one of them has to have its first element transferred to output before revisiting. See Figure 14, where the transitions have been chosen to conform to the revisiting requirements. As each column either has no guard or two complementary guards, no additional rows are needed.

The translation of this table to C++ is given below. As the order of the translations of the columns is immaterial, we have placed them in alphabetic order by label.

```
void mMerge() {
  int u,v;
  typedef enum{S,A,B,C,D,E,F,G,H} State;
  State state = S; // control state
  while(true) {
    switch(state) {
      case A: state = (getR(v))?C:D; break;
      case B: if (getR(v)) {putR(); state = B;}
         else state = H; break;
      case C: if (u <= v) {putL(); state = E;}
         else {putR(); state = A;} break;
      case D: putL(); state = F; break;
      case E: state = getL(u)?C:G; break;
      case F: state = getL(u)?D:H; break;
      case G: putR(); state = B; break;
      case H: return;
      case S: state = getL(u)?A:B; break;
    }
  }
}
```

Figure 10. A C++ function for merging two streams translated from Figure 14.

The reason for developing a code matrix for the merge problem was the desire to avoid the superfluous tests of a function like the `eMerge` listed in Figure 8. To see in how far `mMerge` improves in this respect we have run both functions on the same set of pairs of input streams and counted the calls executed in both merge functions.

Such comparisons are of course dependent on the nature of the input streams. For example, the more equal in length the input streams are, the more favourable for `mMerge`. Accordingly we have used a random-number generator to determine the lengths of the input streams. The input streams themselves are monotonically increasing with random increments.

	getL	getR	putL	putR
eMerge	1756	2691	871	1819
mMerge	872	1821	871	1819
eMerge	1067	830	655	410
mMerge	656	411	655	410
eMerge	3261	735	2894	365
mMerge	2895	366	2894	365
eMerge	1355	1024	844	509
mMerge	845	510	844	509

Each pair of successive lines gives the result of running `eMerge` and `mMerge` on the same pair of input streams. The lengths of the streams are not listed separately, as they are equal to the number of calls to `putL` and `putR` shown in the table.

A merge function needs to make at least one call to `getL` (`getR`) for every element of the left (right) input stream. It can be seen that `mMerge` remains close to this minimum, while `eMerge` does not.

This example is notable in that matrix code yields an unfamiliar, test-optimal algorithm by *default*. Structured programming tends to reduce the number of control states. Matrix code lacks this bias: in its use it is natural to introduce control states as needed to serve as memory for test outcomes.

10. Related work

The following comment has been made on Matrix Code: "*Although it reeks of flow charts, the proposal has some merit*." The comment has some merit: flow charts are indeed closely related to Matrix Code. Flow charts were widely used as an informal programming notation from the early 1950s to 1970. Floyd [8] showed how assertions and verification conditions can prove a flow chart partially correct. Hoare [10] introduced the notation of triples for the verification conditions and cast Floyd's method in the form of inference rules for control structures such as `while ... do ...` and `if ... then ... else ...`

Dijkstra observed that verifying assertions are difficult to find for existing code, so that an attempt at verification is a costly undertaking with an uncertain outcome. He argued [4, 5] that code and correctness should be "developed in parallel". The proposal seems to have found no response, if only for the lack of specifics in the proposal. Given the fact that Dijkstra's proposal was considered unrealistically utopian, and still is, it is interesting to read what seems to be the first treatise [9] on programming in the modern sense, published in 1946. Here programs are expressed in the form of *flow diagrams*. At first sight one might think that these are flow charts under another name. This is not the case: flow diagrams consist of executable code integrated with assertions, with the understanding that a consistent flow diagram proves the correctness of the computations performed by it.

The imperative part of a flow diagram was translated to machine code (this was before the appearance of assemblers). I found no indication in [9] that it was even contemplated to split off the imperative part of the flow diagram. Thus we see that what was a vague proposal [4, 5], and regarded as unrealistically utopian in 1970, was fully worked out in 1946 and may have become a practical reality in 1951 when the IAS machine became operational.

By the time flow charts appeared, the proof part of flow diagrams had been dropped. And apparently forgotten, for Floyd's discovery was published in 1967 and universally acknowledged as such. Floyd's format is rather different, and, in our opinion, preferable to the flow diagrams of [9]. Matrix Code can be regarded as a simplification of Floyd's flow chart annotated with assertions, a simplification made possible by the use of transitions that provide a common generalization of statements and tests. Apt and Schaerf

unify statements and tests in their nondeterministic control structures [1].

Code matrices can be regarded as generalized Finite-State Automata. The control states of code matrices are similar to the states in Finite-State Automata; the data states have no counterpart in FSAs. Data states can contain variables of widely varying types. These can include streams of characters, so that code matrices can simulate FSAs with input and/or output. This possibility makes Matrix Code reminiscent of Dana Scott's proposal [13] to put an end to the proliferation of new variations of FSA by replacing them by programs defined to run on suitably defined machines.

In spite of Scott's injunction, variants of FSA continued to appear. Of special interest in this context are *labeled transition systems* which are used to model and verify reactive systems [2]. Here the set of states is often not finite and there is typically no halt state. Such systems are specified by rules of the form $P \xrightarrow{A} Q$ to indicate the possibility of a transition from state P to state Q accompanied by action A. Mathematically the rules are viewed as a ternary relation containing triples consisting of P, A, and Q. This is of course unobjectionable, but the alternative view of the rules as constituting a matrix indexed by states, containing in this instance A as element indexed by P and Q has the advantage of connecting the theory to that of semilinear programming in the sense of Parker. Another variant of FSA are the *augmented transition networks* used in linguistics [15].

The property that a code matrix is both a set of logical formulas and an executable program is reminiscent of logic programming, especially its aspect of separating logic from control [11]. A special form of logic program corresponding to imperative programs was investigated in [3]. The modification of flow charts by means of binary relations was introduced in [14].

11. Conclusions

In this paper we write programs as matrices with binary relations as elements. These matrices can be regarded as transformations in a generalized vector space, where vectors have assertions about data states as elements. Computations of the programs are characterized by powers of the matrix and verified assertions show up as generalized eigenvectors of the matrix. Such results may be dismissed as frivolous theorizing. It seems to us that they are related to the following practical benefits.

Our motivation was to address the fact that imperative programming is in an unsatisfactory state compared to functional and logic programming. In the latter paradigms, implementation is, or is close to, specification. In imperative programming the relation between implementation and specification is the verification problem, a problem considered too hard for the practising programmer. We proposed Matrix Code as an imperative programming language where the same construct can be read as logical formula and can serve as basis for a routine translation to Java or `C`.

Another practical benefit is that it seems possible in some cases to develop algorithms incrementally by small, obvious steps from the specification. In this paper we go through such steps for an algorithm to fill a table with prime numbers using the method of trial division. Whether or not this success is an exceptional case, it seems certain that progress has been made in the direction of the old dream according to which the production of verified code is facilitated by developing proof and code in parallel.

Acknowledgments

Thanks to Paul McJones, Mantis Cheng, and the reviewers for their help in improving the paper. This research benefited from facilities provided by the University of Victoria and by the Natural Science and Engineering Research Council of Canada.

References

[1] K.R. Apt and A. Schaerf. Search and imperative programming. *POPL '97*, pages 67–79.

[2] C. Baier and J.P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] Keith L. Clark and M.H. van Emden. Consequence Verification of Flowcharts. *IEEE Transactions on Software Engineering*, SE-7:52–60, January 1981.

[4] E.W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.

[5] Edsger W. Dijkstra. Concern for correctness as a guiding principle for program composition. In J.S.J. Hugo, editor, *The Fourth Generation*, pages 359–367. Infotech, Ltd, 1971.

[6] Edsger W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 1–72. Academic Press, 1972.

[7] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[8] Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[9] H.H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. Part II, volume 1, 1946. Reprinted in: *John von Neumann: Collected Works*, Pages 80 - 151, volume V. A.H. Taub, editor. Pergamon Press, 1963.

[10] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[11] R.A. Kowalski. Algorithm = Logic + Control. *Comm. ACM*, 22:424–436, 1979.

[12] D. Stott Parker. Partial order programming. Technical Report CSD-870067, Computer Science Department, University of California at Los Angeles, 1987.

[13] Dana Scott. Some definitional suggestions for automata theory. *Journal of Computer and Systems Sciences*, 1:187–212, 1967.

[14] M.H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, vol. 3(1979), pp 148–159.

[15] W.A. Woods. Transition network grammars. *Comm. ACM*, 13:591–606, 1970.

B:	A:	S: p[0..N-1] exists & N>1	
	k >= N		H: p[0..N-1] contains the first N primes
p[n]*p[n]>j; p[k++]=j		p[0] = 2; p[1] = 3; k = 2	A: p[0..k-1] contains the first k primes & k <= N
	k<N; j = p[k-1]+2; n=0		B: A & k<N & relB(p,k,n,j)

Figure 11. In column A we have added a transition in column A for the case that `k < N`. In that case we can start finding the next prime after `p[k-1]` because we know that there is enough space in `p` to store it. `relB(p,k,n,j)` means that there is no prime between the last prime found and `j` and that `n<k`, and that `j` is not divided by any prime in `p[0..n]`.

C:	B:	A:	S: p[0..N-1] exists & N>1	
		k >= N		H: p[0..N-1] contains the first N primes
	p[n]*p[n]>j; p[k++]=j		p[0] = 2; p[1] = 3; k = 2	A: p[0..k-1] contains the first k primes & k <= N
j%p[n+1]!=0; n++		k<N; j = p[k-1]+2; n=0		B: A & k<N & relB(p,k,n,j)
j%p[n+1]==0; j += 2; n=0	p[n]*p[n]<= j			C: B & p[n]*p[n] <= j

Figure 12. This figure is both a general example of a code matrix and the final stage of the development consisting of the sequence of Figures 5, 6, and 11. Change from Figure 11: row and column with label C are added. There are no incomplete columns. This, as well as each of the previous versions is partially correct, as implied by the validity of the verification condition for each of the null matrix elements. The absence of incomplete columns opens the possibility of total correctness, but does not prove it.

A	S:(?,?)	
		H:(e,e)
	getL(u)	A:(u:?,?)
	!getL(u)	B:(e,?)
getR(v)		C:(u:?,v:?)
!getR(v)		D:(u:?,e)

Figure 13. See Figure 9. An input stream needs to be tested; the left one is chosen arbitrarily. This gives rise to new conditions. Columns for these will cause addition of yet more conditions. See Figure 14.

G	F	E	D	C	B	A	S:(?,?)	
	!getL(u)				!getR(v)			H:(e,e)
				u>v; putR()			getL(u)	A:(u:?,?)
putR()					getR(v); putR()		!getL(u)	B:(e,?)
		getL(u)				getR(v)		C:(u:?,v:?)
	getL(u)					!getR(v)		D:(u:?,e)
				u <= v; putL()				E:(?,v:?)
			putL()					F:(?,e)
		!getL(u)						G:(e,v:?)

Figure 14. The complete code matrix for the merging problem, continuing Figures 9 and 13.